Received 1 October 2018; | Revised 19 February 2019; | Accepted ddmmyyyy

DOI:

RESEARCH ARTICLE

Accelerating Magnetic Induction Tomography Based Imaging through Heterogeneous Parallel Computing

David W. Walker[1] | Stephan C. Kramer[3] | Fabian R. A. Biebl[†1] | Paul D. Ledger[2] | Malcolm Brown[1]

[1]School of Computer Science & Informatics, Cardiff University, Cardiff, UK
[2]Zienkiewicz Centre for Computational Engineering, Swansea University, Swansea, UK
[3]Fraunhofer Institute for Industrial Mathematics, Kaiserslautern, Germany

Correspondence
David W. Walker.
Email: WalkerDW@cardiff.ac.uk

Present Address
[†] Math2Market GmbH, Kaiserslautern, Germany

Summary

Magnetic Induction Tomography (MIT) is a non-invasive imaging technique which has applications in both industrial and clinical settings. In essence, it is capable of reconstructing the electromagnetic parameters of an object from measurements made on its surface. With the exploitation of parallelism it is possible to achieve high quality, inexpensive MIT images for biomedical applications on clinically relevant time scales. In this paper we investigate the performance of different parallel implementations of the forward eddy current problem, which is the main computational component of the inverse problem through which measured voltages are converted into images. We show that a heterogeneous parallel method that exploits multiple CPUs and GPUs can provide a high level of parallel scaling, leading to considerably improved runtimes. We also show how multiple GPUs can be used in conjunction with `deal.II`, a widely-used, open source finite element library.

KEYWORDS:
Computational electromagnetics, magnetic induction tomography, parallel applications

1 | INTRODUCTION

Magnetic Induction Tomography (MIT) is a non-invasive imaging technique which has applications in both industrial and clinical settings. In essence, it is capable of reconstructing the electromagnetic parameters of an object from measurements made on its surface. These parameters are the permittivity, ϵ, the permeability, μ, and the conductivity, σ. An MIT device consists of two sets of coils placed around the boundary of the object to be imaged. The first set of coils is used for the purpose of excitation, and by passing a current through each coil in turn, a primary magnetic field is created. The second set of coils is then used for measurement. This procedure causes an eddy current when each of the primary magnetic fields interacts with a conducting body inducing secondary magnetic fields, and hence voltages, that are measured in the second set of coils. In order to estimate the electromagnetic properties of the material, (ϵ, μ, σ), from the induced currents and measured voltage, an inverse problem must be solved. In many practical applications the distribution of one or more of these material parameters is assumed to be constant throughout the medium of interest.

Conventional imaging techniques for imaging cerebral stroke, such as Magnetic Resonance Imaging (MRI) and Computer Tomography (CT), are expensive. Although MRI may be used for real-time image reconstruction [1], it has been proposed that MIT can offer a low cost alternative in the first stages of diagnosis [2]. However, the low conductivity contrast between biomedical tissues presents significant challenges to MIT, and there are considerable difficulties in employing current computational techniques to solve the associated inverse problem [3,4].

Enabling MIT to take the step from being an experimental technique, which has already received some clinical interest, to become a viable imaging technique for the detection and monitoring of conditions, such as cerebral stroke, requires a step change in the quality of the reconstruction of the passive electromagnetic parameters and, therefore, an improvement of the computational approach used for the solution of the inverse problem. This is not straightforward as the inverse problem is challenging, being severely ill-posed (small changes in measured voltages can imply large changes in ϵ, μ, σ) and suffering from limited noisy measurements. Common approaches include regularised Tikhonov schemes based around Gauss-Newton strategies, such as [5], although a variational algorithm developed by Brown and Jais [6],

based on previous work by Knowles[7], may offer an alternative self-regularising approach. It is well known that the level and type of regularisation plays a crucial role in determining the accuracy of the inverse solution, and hence the image quality. A further important factor is the accuracy of the solution of the direct problem, which provides derivative information and is repeatedly solved to guide the inverse algorithm. This is commonly overlooked due to the level of noise in the measured data.

Our focus here is on the MIT direct problem, and its solution using parallel computing systems containing multicore processors connected to one or more graphical processing units (GPUs). This solution involves the eddy current approximation of Maxwell's equations[8], which must be solved for the electromagnetic fields in the imaging region for a given set of current sources and distributions of μ and σ [a]. We formulate the MIT direct problem according to the regularised eddy current formulation proposed in[9] and employ a finite element discretisation. The vectorial nature of Maxwell's equations necessitates the use of Nédélec finite elements, and the hp–version of these elements using the basis functions proposed by Schöberl and Zaglmayr[10] has been employed. These basis function sets are hierarchic in nature and offer high levels of accuracy, which is appealing in our MIT application. These hp–Nédélec hexahedral finite elements have been implemented in the `deal.II` finite element library[11,12], which provides the computational framework for this project. Following element assembly, a complex symmetric linear system of equations is assembled, which is solved iteratively by a preconditioned Generalized Minimal RESidual (GMRES) iterative solver.

Some vector-valued problems, for instance the mechanical displacements in linear elasticity, can be discretised by simple Lagrangian elements. However, problems arise when Maxwell's equations, expressed in vector wave equation form, are discretised by such elements. Here the continuity requirements at material interfaces dictates that only the tangential component of the electric and magnetic fields be continuous and that the normal component be discontinuous. Application of standard continuous elements would impose too much continuity and result in spurious solutions. Instead, the Nédélec elements provide the correct tangential continuity of the discrete fields and guarantee that the curl of the gradient of a scalar field vanishes even after discretisation. For the standard Lagrange elements this is not the case in general. Issues of multiply-connected domains are also important for eddy current problems and the vector potential A-based formulation we employ here overcomes these issues[9].

To numerically solve the MIT problem we use `deal.II`[11], version 8.5.1, a general-purpose C++ library. It offers an infrastructure for implementing parallelised higher-order finite element solvers on unstructured hexahedral meshes. Parallelisation can be done either via MPI and domain decomposition for large-scale distributed problems or via a shared-memory approach using OpenMP [b] or `deal.II`'s WorkStream framework[13]. WorkStream is a scalable C++-template framework for parallelization of operations involving loops over cells of the mesh, e.g. matrix assembly. It is based on Intel's Threading Building Blocks (TBB) library.

The approximated MIT direct problem is well suited to parallelisation on GPUs, whose costs have been driven down by mass production for the computer games industry. In this paper, the parallelization of the MIT direct problem is investigated, using both multicore CPUs and manycore GPUs. Firstly, two approaches to multicore parallelism are compared: OpenMP and `deal.II`'s WorkStream framework[13]. The WorkStream implementation is then used as the basis of a heterogeneous parallel code using a single GPU and several host cores (of possibly multiple CPUs, if available). Its performance is investigated for a number of different NVidia GPUs. Finally, the code is extended to the general case of multiple CPUs and multiple GPUs.

The WorkStream framework can also be combined with `deal.II`'s MPI parallelisation so that each MPI process may work on a larger subdomain. However, our MIT project aims at off-line interactive simulations, as may be necessary in a hospital, and therefore we do not consider MPI parallelisation here. The MPI approach, combined with OpenMP and the FreeFem++ library, has been investigated by Tournier et al.[14], who have solved the full inverse problem using 1024 cores of the Curie supercomputer. This approach relies on network connectivity between the MIT equipment and the supercomputer, whereas our approach performs most of the computation on a GPU system that can be co-located with the MIT equipment.

In Sec. 2 the mathematical formulation of the relevant computational electromagnetics problem is presented, and the solution method is described. Sec. 3 describes how the problem is parallelised on multicore and manycore platforms. In Sec. 4 performance results are presented and discussed. Finally, in Sec. 5 conclusions and future work are discussed.

2 | MATHEMATICAL FORMULATION

The time harmonic eddy current model describes the behaviour of the electromagnetic field for the MIT direct problem. Let Ω_C denote a conducting object with uniform conductivity and permeability, which is located in an unbounded free space region $\mathbb{R}^3 \setminus \Omega_C$. The conductivity and permeability are given by

$$\sigma = \begin{cases} \sigma_* & \text{in } \Omega_C \\ 0 & \text{in } \mathbb{R}^3 \setminus \Omega_C \end{cases}, \qquad \mu = \begin{cases} \mu_* & \text{in } \Omega_C \\ \mu_0 & \text{in } \mathbb{R}^3 \setminus \Omega_C \end{cases},$$

where $\mu_0 := 4\pi \times 10^{-7}$ H/m. We assume that the conducting body is excited by a divergence-free current source of amplitude $\vec{J}^s$ and frequency ω, located away from Ω_C. Then the interaction magnetic and electric fields, $\vec{H}$ and $\vec{E}$ respectively, satisfy the eddy current

[a] Note that for highly conducting objects and the low frequencies of operation, the displacement current, and hence ϵ, is neglected in the Maxwell system.

[b] See http://www.openmp.org

equations [15]:

$$\text{curl}\,\vec{E} = -\mathrm{i}\omega\mu\vec{H}, \qquad \text{curl}\,\vec{H} = \sigma\vec{E} + \vec{J}^s. \tag{1}$$

The interaction fields decay at appropriate rates as $|\vec{x}| \to \infty$ [15], which computationally allows us to truncate the unbounded domain at a finite distance from the object and defines the bounded domain $\Omega := \Omega_C \cup \Omega_{NC}$, where Ω_{NC} is the truncated part of $\mathbb{R}^3 \setminus \Omega_C$. On the object's surface, $\partial\Omega$, we apply simple zero Dirichlet boundary conditions.

We adopt the $\vec{A}$-based regularised form of the eddy current model described in [9], which has the associated weak form: Find $\vec{A} \in \vec{V} :: \{\vec{A} \in \vec{H}(\text{curl}) : \vec{n} \times \vec{A} = \vec{0}\ \text{on}\ \partial\Omega\}$ such that

$$\int_\Omega \mu_r \text{curl}\,\vec{A} \cdot \text{curl}\,\vec{w}\mathrm{d}\Omega + \int_\Omega \tilde{\kappa}\vec{A} \cdot \vec{w}\mathrm{d}\Omega = \mu_0 \int_{\Omega_{NC}} \vec{J}^s \cdot \vec{w}\mathrm{d}\Omega \qquad \forall \vec{w} \in \vec{V}, \tag{2}$$

where $\mu_r := \mu/\mu_0$, $\vec{A}$ is a vector potential defined as $\vec{B} := \mu\vec{H} = \text{curl}\,\vec{A}$ and is such that $\vec{E} = -\mathrm{i}\omega\vec{A}$ in Ω_C. The parameter $\tilde{\kappa}$ is defined as

$$\tilde{\kappa} := \begin{cases} \mathrm{i}\omega\mu_0\sigma_* & \text{in}\ \Omega_C \\ \mathrm{i}\varepsilon & \text{in}\ \Omega_{NC} \end{cases},$$

where ε is a small regularisation parameter.

We employ an hp-finite element discretisation for the solution of Eq. (2) using the `deal.II` finite element library [11,12]. The `deal.II` finite element library allows for structured and unstructured meshes of hexahedral elements, which is well suited to the MIT direct problem due to the convenience it offers of providing a voxelated grid for approximating $\vec{A}$ as well as for describing σ and μ. The latter is important for future extensions involving the solution of the associated MIT inverse problem for recovering the material coefficients σ and μ.

To correctly discretise Eq. 2 using `deal.II` without the so-called "sign-conflict", the Nédélec finite element has been re-implemented by Kynch and Ledger [12]. Their implementation is based on the Nédélec finite element basis of Schöberl and Zaglmayr [10], which offers the possibility of non-uniform h (mesh) and p (polynomial) refinements. The sign conflict problem is associated with edge and face parametrisation on general hexahedral elements. We have ported the implementation by Kynch and Ledger to GPUs using NVidia's CUDA programming model. Our CUDA version is independent of the particular version of `deal.II`[c]. In the following, we only discuss uniform polynomial degrees and conforming hexahedral meshes to avoid the technical issue of hanging nodes. Local refinement will certainly be crucial for resolving spatial inhomogeneities in the material parameters while keeping the computation times within reasonable limits without sacrificing accuracy. Yet, for benchmarking the performance, in particular of the matrix assembly, this is immaterial as the complexity of the assembly depends on how many cells there are and not on whether they are locally refined. The matrix assembly can be accelerated by calculating only the real-real part of the cell matrices and converting them to the complex-valued matrices actually needed when adding the cell matrices into the global matrices. This is possible because the problem is symmetric and in the context of MIT it is reasonable to assume that the permittivity and conductivity are constant on each cell. Therefore, they can be taken out of the integrals in the cell-wise variational form:

$$\sum_{K\subset\Omega} \mu_{r,K} \int_K \text{curl}\,\vec{A} \cdot \text{curl}\,\vec{w}\mathrm{d}K + \tilde{\kappa}_K \int_K \vec{A} \cdot \vec{w}\mathrm{d}\Omega = \mu_0 \int_K \chi(\Omega_{NC})\vec{J}^s \cdot \vec{w}\mathrm{d}\Omega \qquad \forall \vec{w} \in \vec{V}, \tag{3}$$

where $\mu_{r,K}$ and $\tilde{\kappa}_K$ indicate the values of μ_r and $\tilde{\kappa}$ on cell K, and $\chi(\Omega_{NC})$ is the characteristic function of the non-conducting subdomain. The operation count for the assembly of the cell matrices is proportional to the square of the number of degrees of freedom per cell, hence it should be possible to save a factor of 4 in execution time for higher order finite elements or some memory, if this becomes an issue.

Following the discretisation of Eq. (2), a complex linear system of equations

$$\mathbf{K}x = b \tag{4}$$

is assembled. For its solution, we employ a preconditioned GMRES iterative solver using the block Jacobi preconditioner proposed in [9]. This preconditioner exploits the hierarchic nature of Schöberl and Zaglmayr's Nédélec basis functions, and takes advantage of their special construction, which allows the grouping of gradient and non-gradient basis functions. The treatment of the former reduces to the preconditioning of a simple elliptic operator. The resulting preconditioner has been shown to be computationally robust with respect to $\tilde{\kappa}$ in [9]. In [12], a series of further examples is included to demonstrate the performance of this technique within the `deal.II` computational framework.

3 | PARALLELISATION

All the OpenMP results presented in this paper are for a machine named g00 with two sockets, each containing a 2GHz Intel Xeon E5-2620 processor. This processor has 6 cores with a 256 KB L2 cache per core, and a 15MB L3 cache. The system has a 16 GB main memory and the operating system is Red Hat Enterprise Linux Server release 6.2. Version 4.8.5 of the gcc compiler was used with the "O3" optimization flag set. This supports OpenMP 3.1. In addition, performance results for the following types of NVidia GPU are presented:

1. Tesla K20Xm with compute capability SM3.5, running CUDA 8.0. This GPU has 6 GB of memory and the g00 system described above hosts two of them.

2. GeForce GTX 750 Ti. This GPU has 2 GB of memory and is hosted by an Intel Core i3-3240 running at 3.4 GHz with 2 cores and a 3 MB cache.

3. GeForce GTX 580. This GPU has 1.5 GB of memory and is hosted by an Intel Xeon E5620 running at 2.4 GHz with 4 cores and a 12 MB cache.

[c] We use `deal.II` version 8.5.1. However, the finite elements implemented in plain C++ by Kynch and Ledger are part of `deal.II` since version 9.1

4. Tesla P100. This GPU has 16 GB of memory and is hosted at the University of Göttingen by an Intel Xeon E5-2609 running at 2.4 GHz with 4 cores and a 10 MB cache. Each node of the Hawk cluster described below also contains two Tesla P100 GPUs.

5. Tesla K80. This GPU has 24 GB of memory and is hosted by an Intel Xeon E5-2609 v4 processor running at 1.7 GHz with 8 cores and a 20 MB cache.

6. GeForce GTX 1080. This GPU has 8 GB of memory and 2560 processing cores. It is hosted by the P600 system containing two Intel Xeon E5-2687W v4 processors running at 3.00 GHz, each with with 12 cores and a 30 MB cache.

In addition, performance results on multicore systems with up to four GPUs are presented to demonstrate scalability with respect to the number of GPUs used. The systems used in this part of the work are:

1. P500: a system with two CPU sockets and four NVidia Quadro P2000 in PCIe-3.0x16 slots. Each CPU socket is occupied by one Intel Xeon E5-2660 CPUs with 20 MB of cache and 8 cores running at 2.20 GHz. The CPUs have access to 128 GB of memory. The software environment is Ubuntu 17.10 and CUDA 9.2.

2. Hawk: a GPU node of the Hawk cluster. The Hawk cluster consists of 201 nodes, totalling 8040 cores and 46.08 TB of memory. Each node contains two Intel Xeon Gold 6148 processors running at 2.40 GHz with 20 cores each. Standard nodes have 192 GB of memory and GPU-enabled nodes have 384 GB. Each GPU node contains two P100 GPUs.

For both these systems the gcc-7.3.0 compiler was used with default flags from the qmake release mode. QtCreator 4.3 and Qt 5.9 were used as a development environment.

All timing results are the average over eight executions. The standard deviation was also calculated, but is not shown in the plots as it was always less than 2% of the average value.

The problem considered is the "sphere benchmark", which provides an approximation to the shape of a human head. The sphere benchmark is defined as problem 6 in [16], consisting of a conducting sphere in an unbounded region of free space with a uniform background field of magnetic flux density $\vec{B}_0 = \mu_0 \vec{H}_0 = (0, 0, 1)^T$ and angular frequency $\omega = 100\pi$ radians/sec. The sphere has radius $R = 0.05$ m and material parameters $\sigma_* = 10^7$ S/m and $\mu_* = 20\mu_0$. The tolerance used by GMRES in the solution of Eq. (4) is 10^{-7}. The benchmark employs an initial mesh consisting of 19 cells, which may then be changed by refinement. Other numerical parameters are the Nédélec degree p for representing $\vec{A}$, and the polynomial order m of the interpolation of the boundary of the geometry.

It should be noted that the numerical solutions of the eddy current problem are independent of the implementation and hence are identical for the sequential, shared memory, and CUDA versions.

3.1 | Profiling the Sequential Code

Before parallelizing the forward problem the code was first profiled to see where parallelization was likely to be most effective. Two cases were considered:

1. Mapping order $m = p + 1$, and the quadrature order, $q = p + 2$. This is the isoparametric case in which the polynomial degree for approximating the domain boundary is the same as the degree of the finite elements. The results are shown in Figs. 1 and 2 .

2. Mapping order $m = 2$, and the quadrature order, $q = 2p+3$. This will be referred to as the non-isoparametric case. The results are shown in Figs. 3 and 4 .

It should be noted that the quadrature order is the number of quadrature points per dimension, so $n_q = q^3$ for this problem; the finite element degree is $p + 1$, where p is the Nédélec degree.

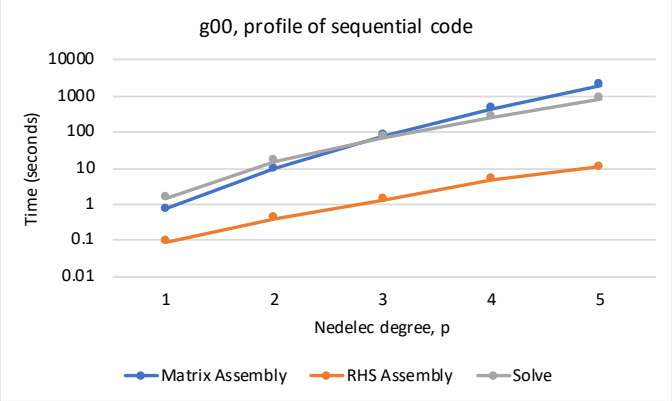

FIGURE 1 Time for main computational phases of the sequential code for the isoparametric case.

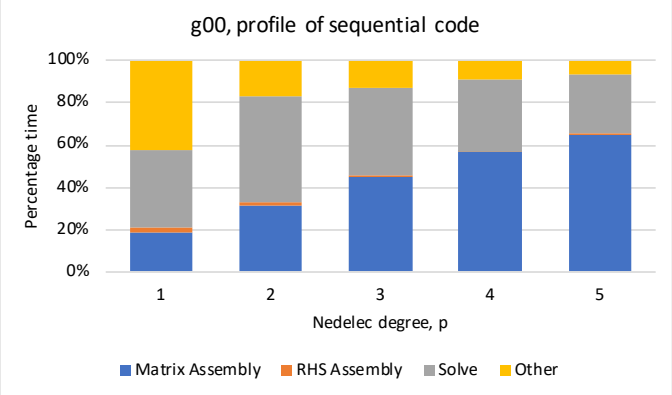

FIGURE 2 Relative time spent on each phase of the sequential code for the isoparameteric case.

The figures show that the main computational phases of the problem are matrix assembly, right-hand side assembly, and the solution of

Eq. (4). The remainder of the execution time is taken up mainly with mesh generation, input, and output. For both cases the matrix assembly phase dominates the execution time for $p \geq 2$, and becomes increasingly dominant as the Nédélec degree, p, increases. It was, therefore, decided to focus the parallelization effort on the matrix assembly phase. It is also apparent that for the largest value of $p = 5$ the execution time is 2.7 hours for the isoparametric case, and almost one day for the non-isoparametric case, which is too long to be useful in a clinical context. Thus, parallelization is needed to reduce the execution time to make the use of MIT for medical imaging more practicable.

The time for the matrix assembly phase of the sphere benchmark depends on the number of quadrature points per cell, n_{q}, and the number of degrees of freedom per cell, n_{dof}, which is given by [17]:

$$n_{\mathrm{dof}} = 3(p+1)(p+2)^2$$

A linear least squares fit of the timings for the matrix assembly phase shown in Fig. 3 gives a best-fit time in seconds of:

$$T(p) = 8.14 \times 10^{-6} n_{\mathrm{dof}}^2 (n_{\mathrm{q}} + 39.97) \quad (5)$$

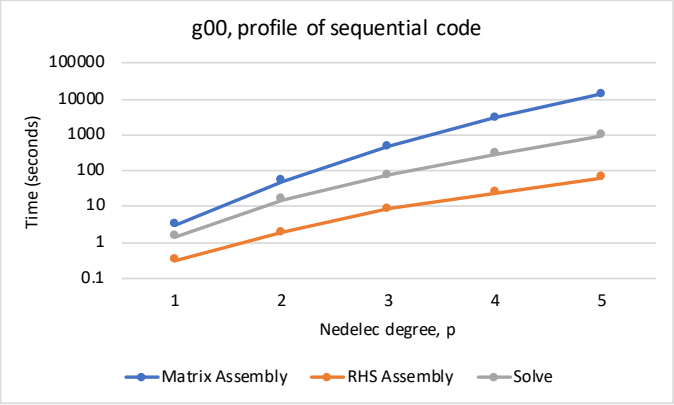

FIGURE 3 Time for main computational phases of the sequential code for the non-isoparameteric case.

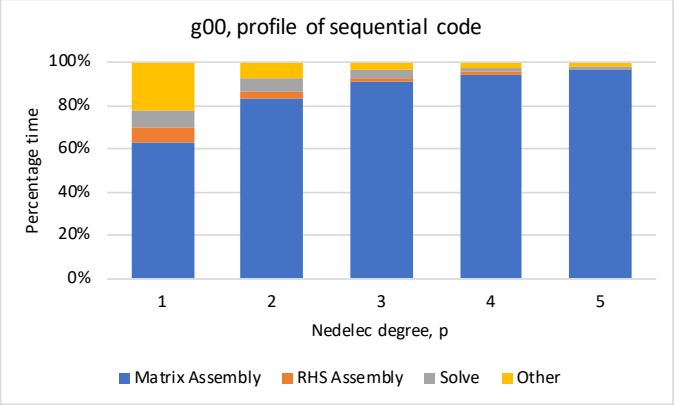

FIGURE 4 Relative time spent on each phase of the sequential code for the non-isoparameteric case.

with a correlation coefficient of $r = 0.9984$. The form of Eq. (5) is as expected from the loop structure of the matrix assembly code, as explained in Sec. 3.2.

3.2 | Multicore Parallelism with OpenMP

The structure of the nested loop that dominates the computation in the matrix assembly phase is shown in Alg. 1. The loops over i and j give rise to the n_{dof}^2 term in Eq. (5), and the inner loop to the n_q term. The number of degrees of freedom per cell ranges from about 100 up to over 1000 as p varies from 1 to 5, so it was decided to parallelize the algorithm with OpenMP just over the i loop. This will expose sufficient parallelism for execution on a few multicore processors. In addition, the range of the j loop depends on the loop index of the i loop and there is no standard way in OpenMP of parallelizing over a doubly nested loop of this type.

Algorithm 1 Loop structure of matrix assembly phase.

```
foreach active cell do
    for ( i = 0; i < n_dof; i++ ) {
        for ( j = i; j < n_dof; j++ ) {
            ⋮
            for ( k = 0; k < n_q; k++ ) {
                Accumulate contributions to element (i,j) of cell matrix
            }
            ⋮
        }
    }
    Add contributions from cell matrix to global stiffness matrix
end
```

3.3 | Multicore Parallelism with the WorkStream Framework

In Sec. 3.2, parallelism is exploited by distributing the outer loop over the degrees of freedom of a cell to a set of threads, which are then scheduled by the OpenMP runtime. The threads evaluate the contributions of a cell to the cell matrix, which are then added into the global stiffness matrix by the master thread. An alternative approach to the assembly of the global stiffness matrix is to make use of `deal.II`'s `WorkStream` framework, which is based on Intel's Threading Building Blocks (TBB) library [18], for an efficient shared memory parallelization. This distributes the assembly of the cell matrices over several worker threads that put their results into a circular buffer. Each thread is responsible for a set of cells, so the parallelization is at the outer level in Alg. 1. This is sketched in Alg. 2. To avoid unnecessary synchronization there is exactly one consumer thread. It retrieves cell matrices from the buffer and adds their elements to the global stiffness matrix. The complexity of computing a cell matrix is $\mathcal{O}(n_{\mathrm{dof}}^2 n_{\mathrm{q}})$ while adding the cell matrix to the global stiffness matrix is only $\mathcal{O}(n_{\mathrm{dof}}^2)$. Taking into account the number of

cells n_{cells} the total costs for matrix assembly is $\mathcal{O}(n_{\text{cells}} n_{\text{dof}}^2 n_{\text{q}})$ while solving an (elliptic) problem takes only $\mathcal{O}(n_{\text{cells}} n_{\text{dof}})$ in the optimal case. Since the number of quadrature points grows with the polynomial degree of the finite element functions this is effectively the competition between cubic and quadratic complexity, i.e. for high orders it is the computation of the cell matrices which will dominate the run time.

Algorithm 2 Loop structure of parallelized assembly phase of a symmetric matrix on the CPU. Each worker thread computes the cell matrices for a subset of cells and stores them in a circular buffer. The first step for each cell is the evaluation of the needed FE data on the real cell, i.e., shape values and/or gradients (symbolized by the matrix $\mathbf{U}$), quadrature weights times the Jacobian of the reference-to-real cell mapping times the value of the PDE coefficient at the quadrature point (summarized as the diagonal matrix $\mathbf{D}$). The actual assembly of the cell matrix $\mathbf{K}^{\text{cell}}$ is a matrix-matrix-matrix product. Finally, the consumer thread takes the cell matrices from the circular buffer and adds the entries of the cell matrices to the global stiffness matrix $\mathbf{K}$.

```
foreach thread do
    if thread.type == worker then
        for ( cell ∈ {cells assigned to thread} ) {
            [U,D] = fe_values.reinit(cell)
            K^cell = U · D · U^T
            K^cell.push_to(circular_buffer)
        }
    end
    if thread.type == consumer then
        for ( cell ∈ circular_buffer ) {
            K += K^cell
        }
    end
end
```

Each thread assembles the cell matrices for only a subset of cells and thus this approach can be considered as a kind of domain decomposition, although each thread still has access to the whole triangulation, stiffness matrix and right-hand side.

3.4 | GPU Parallelism with CUDA

In order to port the forward solution of the MIT simulation to CUDA one has to recall a few basic facts about `deal.II`. First of all, due to its object-oriented nature, the port to CUDA can be split into several independent tasks. In particular, the linear algebra classes are completely independent from the problem of assembling the stiffness matrix and right-hand side, i.e., from the implementation of the finite elements and their evaluation on a cell. Porting `deal.II`'s linear solvers to the GPU basically amounts to passing a GPU-capable vector class as a template parameter that fulfills the interface implicitly defined by the way the vector type is used inside the solver classes. For further discussion of this issue the reader is referred to our paper [19].

As shown in Sec. 3.1, the execution time is dominated by the assembly of the global stiffness matrix, particularly for higher order finite elements. As the most costly operation in assembling a cell matrix is the matrix-matrix-matrix product, $\mathbf{UDU}^T$, a further speedup may be gained by offloading this data-parallel computation to one or more GPUs. The computation of one matrix-matrix product in the cell matrix assembly does not fully utilize one GPU, and therefore several host threads can share one GPU without performance losses as GPUs are able to execute several kernels concurrently. To minimize the amount of CPU-to-GPU copy operations the finite element data is pre-computed on the GPU, so it is necessary to copy only the quadrature points on each cell from the CPU to the GPU. In the case of the Nédélec elements the orientations of the edges and faces must also be copied to the GPU. Only then can the evaluation of the shape functions and their derivatives be completed. To compute the values of the shape functions and their derivatives on a GPU it is necessary to port `deal.II`'s finite element framework either to CUDA (or to an alternative such as OpenCL or OpenACC). `deal.II`'s finite element framework consists of two parts:

1. The classes that describe the finite elements,
2. The classes that take care of evaluating a finite element and mapping it to a real cell.

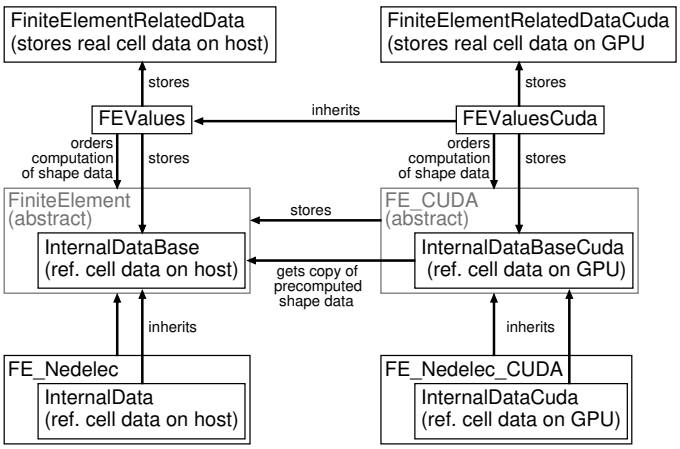

FIGURE 5 Dependency graph of the finite element classes in `deal.II` since version 8.4 (left side), and our classes for extending `deal.II`'s finite element framework to CUDA.

In `deal.II` all finite element classes describing the polynomial space are derived from `FiniteElement`, which defines their common interface. Using this general interface the classes derived from `FEValuesBase` manage the evaluation of a given finite element and its derivatives on a real cell. Their interdependence, and how they have been extended to CUDA, is illustrated in Fig. 5 .

The type of finite element determines how much data can be pre-computed in advance on the reference cell. For instance, the standard Lagrange elements allow the pre-computation of the function values of the polynomials at the given quadrature points and only need to recompute derivatives for each real cell. For Nédélec elements, on the other hand, not even the shape values can be completely pre-computed on the reference cell due to the sign problem [12].

For the CUDA implementation, we define a class structure following that of `deal.II`. The resulting class diagram is asymmetric as we reuse those data from the CPU implementation that have to be computed only once, e.g., the shape values of the Lagrange elements as mentioned above. Analogously to `dealii::FiniteElement` we define a new base class `FiniteElementCuda` that defines the interface for all CUDA-based FE classes. Since it is desirable to re-use the already existing CPU implementation for data that has to be computed only once, the new base class has to store a pointer to its CPU counterpart. Using our re-write of the FE classes, the matrix assembly only requires the insertion of a few copy operations on the host side. This is sketched in Alg. 3.

Algorithm 3 Parallelized assembly phase of a symmetric matrix on the GPU. Now, the circular buffer stores matrix objects whose data is stored on the GPU. The additional step is the copy operation the consumer thread has to perform in order to transfer the final cell matrices to the CPU.

foreach *thread* **do**
  if *thread.type == worker* **then**
    for (*cell* $\in$ *{cells}* **)** {
      $[\mathbf{U}_{gpu}, \mathbf{D}_{gpu}]$ = fe_values_cuda.reinit(cell)
      $\mathbf{A}^{\text{cell}}_{gpu} = \mathbf{U}_{gpu} \cdot \mathbf{D}_{gpu} \cdot \mathbf{U}^{T}_{gpu}$
      $\mathbf{A}^{\text{cell}}_{gpu}$.push_to(circular_buffer)
    }
    if *thread.type == consumer* **then**
      for (*cell* $\in$ *circular_buffer* **)** {
        $\mathbf{A}^{\text{cell}} = \mathbf{A}^{\text{cell}}_{gpu}$
        $\mathbf{A} += \mathbf{A}^{\text{cell}}$
      }
    end
  end
end

The values and derivatives of the finite-element shape functions on the host are managed by the `dealii::FEValues` class. Consequently, we define a class `FEValuesCuda` that inherits `dealii::FEValues` so that it is capable of handling the evaluation of shape functions on the host as well as on the GPU. This entails some additions, which will now be described.

First of all, during its initialization, `dealii::FEValues` calls the function `get_data` of the finite element it is supposed to manage. Thereby, it receives the reference cell shape values and derivatives from the finite element in the `FE_Nedelec::InternalData` structure. Our new class `FEValuesCuda` exploits this mechanism. Its base class `dealii::FEValues` computes the values and derivatives of the shape functions with respect to the reference cell on the CPU, and then copies the result to the GPU, since the initialization is done only once and is thus not time-critical. The corresponding data are contained in a structure `InternalDataCuda`, which is visible on the host side but internally stores data in GPU memory.

During the cell assembly, `dealii::FEValues` calls the `fill_fe_values` function (or a similar function) of its finite element in order to get the values and derivatives of the shape function on the real cell. These are stored in the structure `dealii::internal::FEValues::FiniteElementRelatedData`. Thereby, the finite element is provided with the instance of `FE_Nedelec::InternalData` that was previously initialized. In addition, in the CUDA case, the finite element gets the `FE_NedelecCuda::InternalDataCuda` and an instance of the class `FiniteElementRelatedDataCuda`. The latter is the GPU counterpart of `dealii::internal::FEValues::FiniteElementRelatedData`. This is necessary, because the Nédélec element cannot fully compute the reference cell shapes, since the orientations of the edges and faces of neighboring cells have to be matched, i.e., the edge- and face-related shape values from the reference cell are multiplied by ± 1 depending on the relative orientation of a cell's edges and faces with respect to its neighboring cells. Hence, the Nédélec element (or other finite elements) may need to use information stored on the host, and is allowed to decide what to calculate on the CPU or on the GPU, and what to copy from the host to the GPU. The goal is to calculate as much as possible on the GPU, while trying to limit the copying of data between host and GPU. Finally, the shape values and derivatives are stored on the GPU, and can be used for the cell assembly using CUDA. From a mathematical point of view, for vector-valued PDEs the assembly of a cell matrix amounts to forming the products of higher-order tensors. As tensors can be unrolled into block matrices, the matrix-matrix products from the BLAS standard suffice.

4 | RESULTS

4.1 | Multicore Results

The performance of the OpenMP code on the g00 system was investigated for up to 12 threads and for a range of chunk sizes for guided, dynamic, and static scheduling. Piece-wise linear boundaries were used. For dynamic scheduling it was found that the best performance was for a chunk size of 5 for both the isoparametric and non-isoparametric cases. Performance depends less on chunk size for larger values of the Nédélec degree, p, because as p increases so does the number of degrees of freedom and the degree of parallelism. Similar results were found for

guided [d] and static scheduling, and hence we present here only results for static scheduling with a chunk size of 5. We further restrict our attention to the isoparametric case since this is of most practical interest. Performance results are shown in Figs. 6 and 7 . As expected, the parallel efficiency is highest for the largest problem size, $p = 5$, for which an efficiency of approximately 55% is achieved with 12 threads.

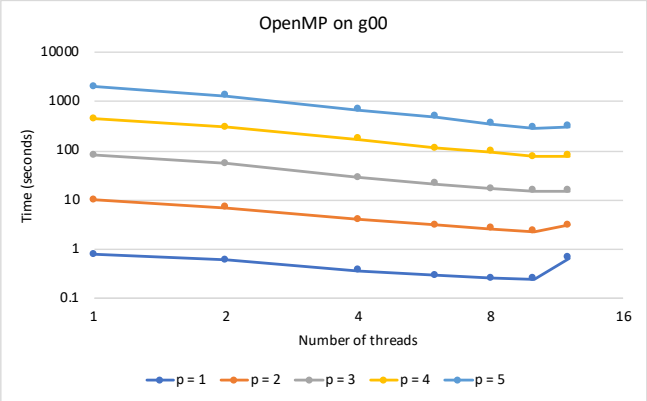

FIGURE 6 Execution time of the OpenMP code for matrix assembly as a function of the number of threads for different Nédélec degree, p. The results shown are for the isoparametric case for static scheduling and a chunk size of 5. The number of cells is 152.

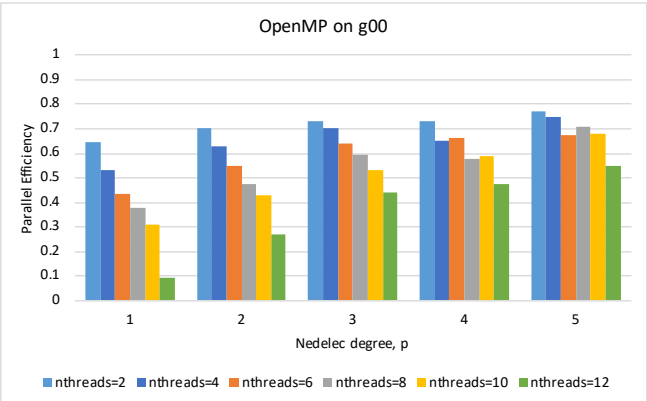

FIGURE 7 Parallel efficiency of the OpenMP code for matrix assembly as a function of Nédélec degree, p, for different numbers of threads. The results shown are for the isoparametric case for static scheduling and a chunk size of 5. The number of cells is 152.

Performance results for the WorkStream implementation described in Sec. 3.3 are shown in Figs. 8 and 9 . These figures show the matrix assembly time for 152 cells as in the OpenMP results in Fig. 6 , and also for the 1216 cells produced by one further level of mesh refinement. Figures 8 and 9 show results for the following two versions of the WorkStream implementation:

- WS1: This WorkStream implementation closely follows the original sequential and OpenMP implementations.
- WS2: This WorkStream implementation differs from WS1 by computing on each cell the curl-curl and mass matrices from real-valued Nédélec elements, and then using these to obtain the final complex-valued cell matrix (see Eq. 3). In the WS1 implementation complex values are used throughout the computing of the cell matrices.

The WorkStream framework of `deal.II` has two main parameters: `chunk_size` and `queue_length`. For the single-GPU results we have used the default values, i.e., `queue_length` is twice the number of threads and `chunk_size`$= 8$. For the multi-GPU benchmarks we used `chunk_size`$= 4$ as this leads to a more equal distribution of work among the GPUs.

Figures 8 and 9 show results for piece-wise linear boundaries and Nédélec degree $p = 1$, 3, and 5, although we have also performed timings for $p = 0$, 2, and 4 that show the same general trends. The WorkStream results show that for smaller problems (small Nédélec degree and/or number of cells) the scope for effective parallelisation is limited. For example, for $p = 1$ and 152 cells no performance gain arises from using more than 4 threads for the WS1 implementation. It is also apparent from the figures that the WS2 implementation is faster than the WS1 implementation, with performance improvements of up to 55%. It was not possible to run the case for $p = 5$ and 1216 cells on g00 due to lack of memory for the sparcity pattern.

Comparison between the OpenMP timings in Fig. 6 and the WorkStream timings in Figs. 8 and 9 shows the OpenMP implementation is slower than both the WS1 and WS2 implementations by factors of up to 2. This is due to the differences in the granularity of parallelisation of the OpenMP and WorkStream implementations. As shown in Alg. 1, for the OpenMP implementation the outer loop over cells is handled by the master thread and parallelism is exploited only at the next loop level. Thus, there is overhead incurred in spawning threads within the loop over cells, although race conditions are avoided when adding the cell matrices into the global system matrix as this is done by the master thread. For the WorkStream implementation, shown in Alg. 2, parallelisation is over the outer cell loop, with each thread handling a set of cells. Each thread gets a larger chunk of work to do than in the OpenMP implementation and there is less overhead in creating threads. A single thread is responsible for adding the cell matrices into the global system matrix thereby avoiding any race condition.

Figures 10 and 11 show the times for matrix assembly on the P600 system for up to 24 threads. As expected, the performance is better than for g00, although the same general trends are exhibited.

4.2 | GPU Performance results

This section presents performance results for the matrix assembly phase for GPU systems in which:

1. One core of a multicore system submits work to a single GPU.

[d] Note that for guided scheduling the chunk parameter gives the minimum chunk size.

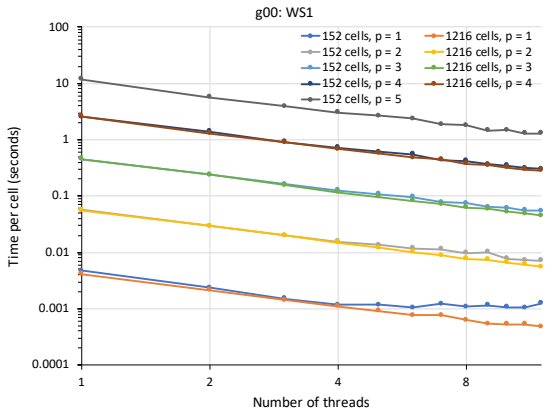

FIGURE 8 Execution times of the WS1 implementation on the g00 system as a function of the number of host threads.

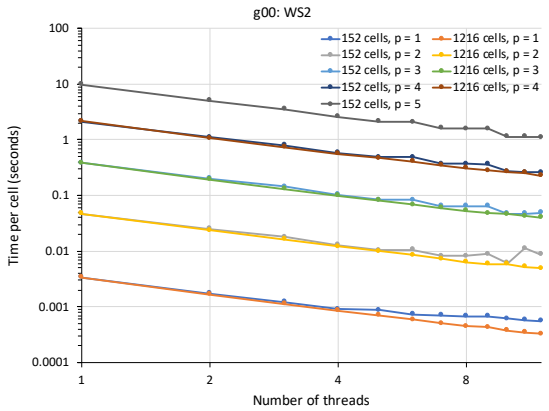

FIGURE 9 Execution times of the WS2 implementation on the g00 system as a function of the number of host threads.

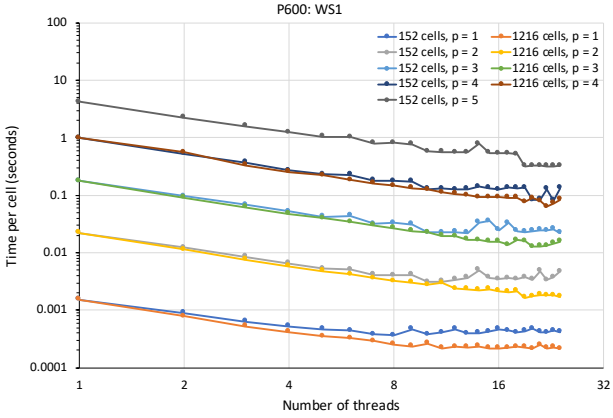

FIGURE 10 Execution times of the WS1 implementation on the P600 system as a function of the number of host threads.

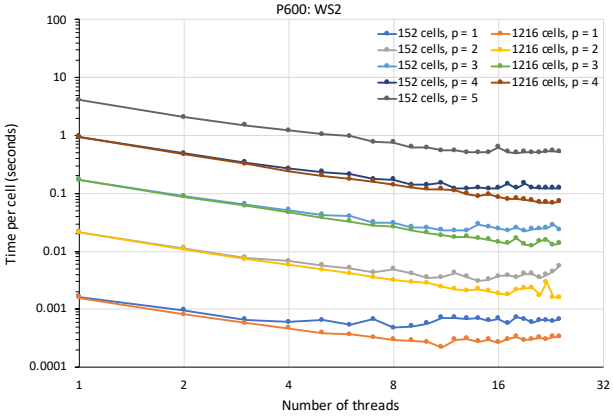

FIGURE 11 Execution times of the WS2 implementation on the P600 system as a function of the number of host threads.

2. Multiple cores submit work to a single GPU.
3. Multiple cores submit work to multiple GPUs.

4.2.1 | Performance Results for a Single CPU and GPU

Figures 12 -15 show the execution time for the isoparametric case with 152 cells for the GTX 750 Ti, GTX 580, Tesla K80, and Tesla P100 GPUs, and their respective hosts. The total execution times are shown, together with:

1. The time to evaluate the values and the curl of the trial functions and the quadrature weights of the Jacobian at all quadrature points (FeValues).
2. The time to perform the triple matrix product, $\mathbf{U}\mathbf{D}\mathbf{U}^{\mathbf{T}}$ (MMM). This includes the time to transfer the necessary data between the host and the GPU.

As expected, the timings in Figs. 12 -15 confirm that the triple matrix product dominates the execution time for the sphere benchmark.

Figure 16 shows the total times for the complete code on the GPUs and on the host computers, while Fig. 17 shows the corresponding speed-up values. For compute-intensive problems (larger p) the newer P100 GPU is substantially faster than the other GPUs used. For all the GPUs, for $p \geq 1$ large speedups are achieved, and for the P100 the speedup up when $p = 6$ exceeds 1000, which validates the approach taken in parallelising the code using `deal.II`.

4.2.2 | Performance Results for Multiple CPUs and a Single GPU.

The WS2 code, in which each host thread is responsible for a set of cells, has been extended as described in Sec. 3.4 to give a CUDA implementation that performs the matrix triple product, $\mathbf{U}\mathbf{D}\mathbf{U}^{\mathbf{T}}$, on a single GPU. The GPU which is supplied with work by multiple host threads. Figures 18 and 19 show the matrix assembly times for a single GPU of the g00 and P600 systems, respectively, for 152 and 1216 cells. Once again, piece-wise linear boundaries are used.

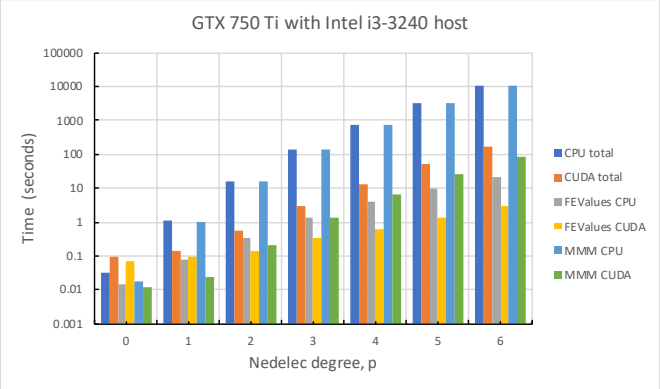

FIGURE 12 Isoparametric case: execution time of the CUDA and CPU codes as a function of the Nédélec degree, p, for an Intel i3-3240 host and GTX 750 Ti GPU.

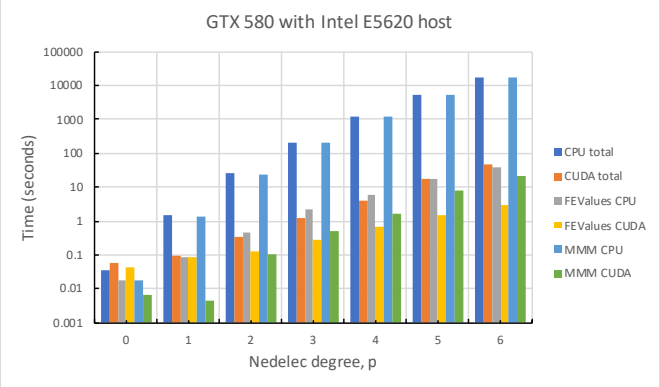

FIGURE 13 Isoparametric case: execution time of the CUDA and CPU codes as a function of the Nédélec degree, p, for an Intel E5620 host and GTX 580 GPU.

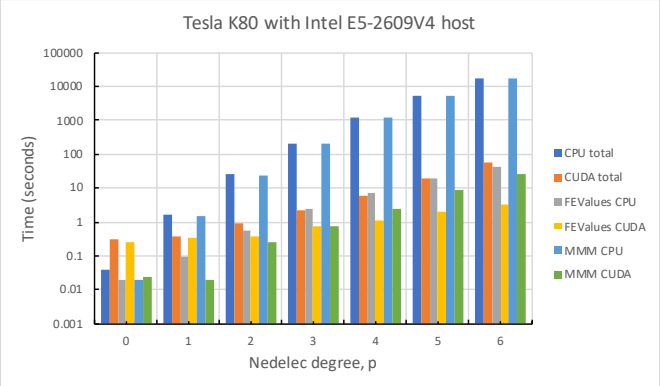

FIGURE 14 Isoparametric case: execution time of the CUDA and CPU codes as a function of the Nédélec degree, p, for an Intel E5-2609V4 host and Tesla K80 GPU.

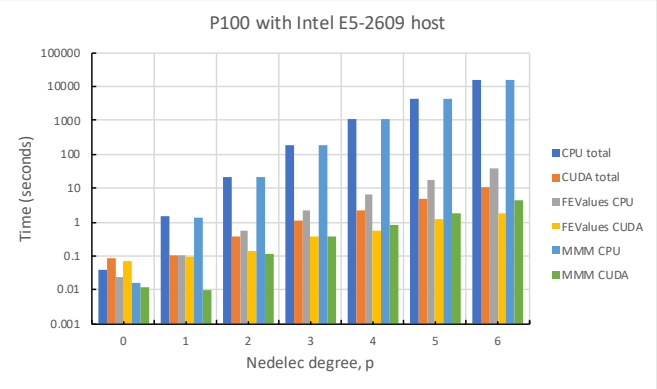

FIGURE 15 Isoparametric case: execution time of the CUDA and CPU codes as a function of the Nédélec degree, p, for an Intel E5-2609 host and P100 GPU.

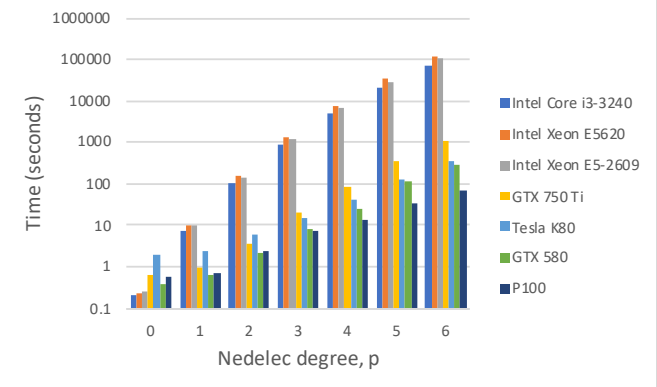

FIGURE 16 Execution times of the CUDA and CPU codes as a function of the Nédélec degree, p, for the complete code in the isoparametric case.

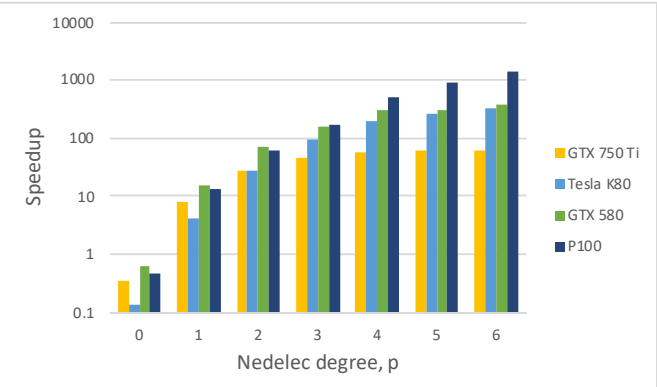

FIGURE 17 Speedup as a function of Nédélec degree, p in the isoparametric case. These values equal the time for the CPU code divided by the time for the CUDA code running on the associated GPU.

When the Nédélec degree, p, is 1 there is no advantage in using the GPU because the amount of computational work done by the GPU is not enough to amortize the time taken to move data between the host and the GPU. For $p \geq 3$ the GPU-enabled code runs faster than the WS1 and WS2 implementations, giving quite substantial speedups in some cases. For example, when $p = 5$ and the number of cells is 152, the

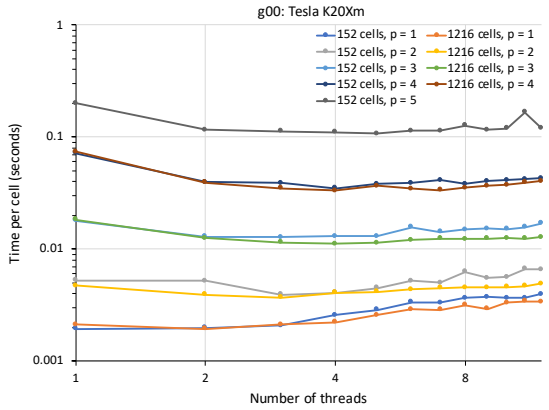

FIGURE 18 Execution times of the CUDA implementation on one GPU of the g00 system as a function of the number of host threads.

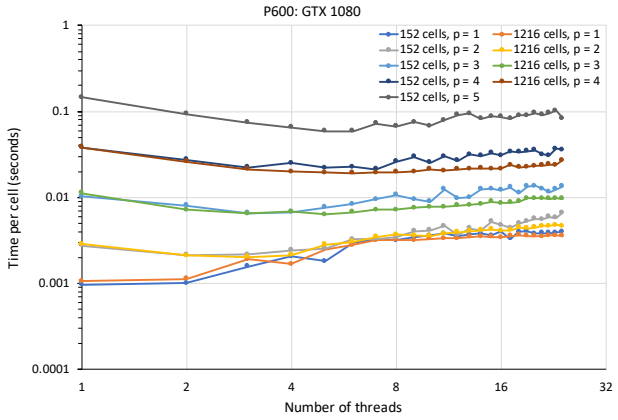

FIGURE 19 Execution times of the CUDA implementation on one GPU of the P600 system as a function of the number of host threads.

speedup is 45.5 and 22.3, respectively, on the g00 and P600 systems when one thread is used. As the number of threads feeding work to the GPU increases the timings decrease at first but then begin to rise or remain approximately constant. As the number of threads increases we expect the rate at which work is sent to the GPU to also increase, which accounts for the initial fall in the execution time. However, as the number of threads increases further the GPU reaches a point at which it is unable to execute the work received any faster.

4.2.3 | Performance Results for Multiple CPUs and Multiple GPUs

To investigate whether using more than one GPU results in better performance we have further extended the code to give an implementation in which multiple threads of a multicore machine feed work to multiple GPUs. Results for the P500 system when four GPUs and multiple CPU threads are used are shown in Fig. 20 , and corresponding results for the WS2 multicore implementation are shown in Fig. 21 . For Nédélec degree $p \leq 4$ we show results for 1216 cells and 9728 cells, respectively. For higher degrees and 9728 cells the sparsity pattern of the global stiffness matrix does not fit into memory of the host (128 GB).

Figures 22 , 23 , and 24 present results for one node of the Hawk cluster in which timings are shown for a single GPU, two GPUs, and the WS2 implementation. The Hawk node contains two P100 GPUs, which are more powerful than the Quadro P2000 GPUs of the P500 system (single precision floating-point peak performance is 3 TFlop/s and 11 Tflop/s for the Quadro P2000 and Tesla P100, respectively). However, while on Tesla P100 the double-precision performance is 50% of the single-precision value, on the Quadro P2000 the double-precision performance is only 3% of the single-precision value.

The main results are:

- Further acceleration of the matrix assembly by moving the matrix-matrix-matrix product to the GPU requires a minimal number of cells which is higher for lower Nédélec degree. An example is this behavior for Nédélec degree $p = 1$ can be seen in Fig. 20 .
- The CPU-only version scales well until all physical cores are occupied. Increasing the number of cells smooths the data, but otherwise has little effect on the wallclock time needed for computing a cell matrix.
- The speedup due to CUDA is multiplicative as long as the number of host threads is less than or equal to twice the number of GPUs. This reflects the fact that GPUs are able to concurrently compute and copy data.
- The additional CUDA parallelization scales with respect to the number of GPUs, i.e., doubling the number of GPUs reduces the assembly time roughly by a factor of 2 until there are twice as many host threads as there are GPUs, i.e., GPU computing and CPU-GPU memory transfers are concurrent
- In its current state the WorkStream+CUDA implementation is inefficient for low polynomial orders and low numbers of mesh cells. The reason is the low computational work per cell in this case and frequent memory transfers of only tiny amounts of data between GPU and CPU. Nevertheless, the higher the Nédélec degree the more CPU cores are needed for the CPU-only version to achieve the same performance as the hybrid version. This could be addressed by batching memory transfers and computation of cell matrices, but this is beyond the scope of this paper.

5 | SUMMARY AND CONCLUSIONS

The research presented here has shown that:

1. On multicore systems `deal.II`'s WorkStream approach to parallelisation usually results in faster execution times compared with the use of OpenMP. This is because the WorkStream approach

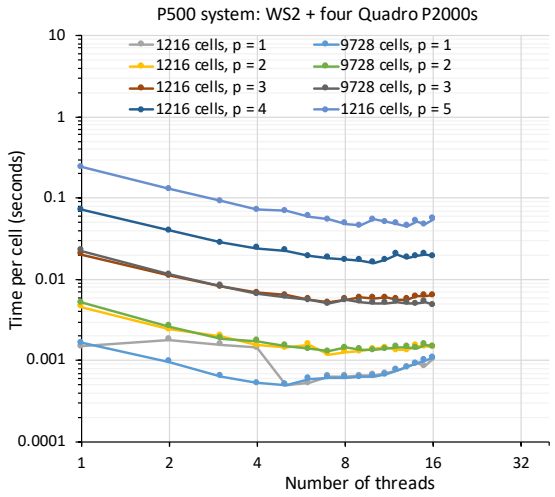

FIGURE 20 Execution times of the CUDA code on the P500 system for four GPUs as a function of the number of host threads.

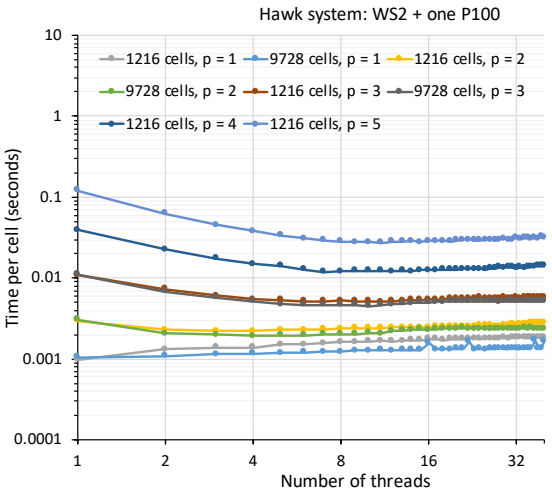

FIGURE 22 Execution times of the CUDA code on the Hawk system for a single GPU as a function of the number of host threads.

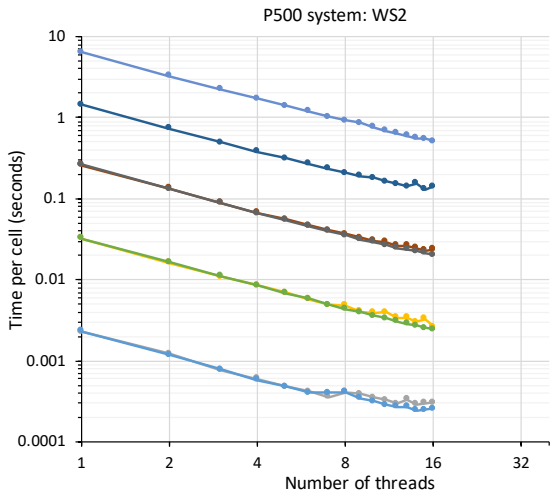

FIGURE 21 Execution times of the WS2 implementation on the P500 system as a function of the number of host threads. The legend is the same as in Fig. 20

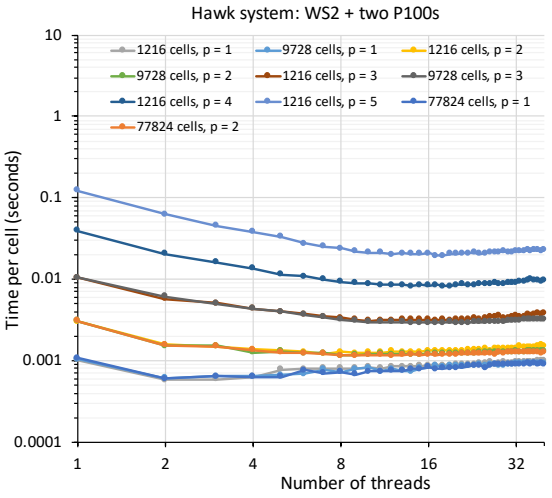

FIGURE 23 Execution times of the CUDA code on the Hawk system for two GPUs as a function of the number of host threads.

exploits parallelism at a higher level of granularity and avoids much of the thread overhead incurred in our naive OpenMP implementation.

2. GPU parallelisation results in substantial reductions in execution time across a wide variety of GPUs, particularly on the P100 system. Even on relatively inexpensive GPUs the performance gains are impressive for problems of interest ($p \geq 2$). This supports the view that GPUs are capable of reducing execution times for MIT imaging to clinically relevant values.

3. The use of multiple NVidia GPUs, using CUDA, can be integrated into the `deal.II` library, resulting in a heterogeneous parallel implementation using multiple CPUs and GPUs.

Future work will investigate the performance and scaling of our code on larger GPU clusters. For problems with large Nédélec degree and a large number of cells the pre-computation of the sparsity pattern may take longer than the matrix assembly, so future work will attempt to reduce the time to find the sparsity pattern and to parallelise it. Further opportunities for optimization of the GPU kernels will also be investigated, in particular the efficient use of resources such as registers to improve thread occupancy. Furthermore, we shall apply the techniques used to produce the multi-GPU version of the code to other areas, such as additive manufacturing. We shall also consider the use of the CUDA CUBLAS library within `deal.II` as a way of further reducing execution times.

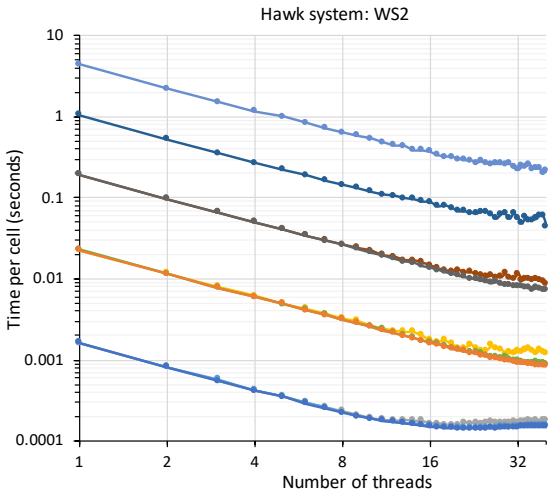

FIGURE 24 Execution times of the WS2 implementation on the Hawk system as a function of the number of host threads. The legend is the same as in Fig. 23

ACKNOWLEDGEMENTS

This research was partially supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grant EP/K024078/1: Inverse Problems for Magnetic Induction Tomography. We are also grateful to the University of Göttingen for access to their GPU systems, and to Dr. Ross Kynch who wrote the sequential code that served as a basis of our parallel codes. We also acknowledge the support of the Supercomputing Wales project, which is part-funded by the European Regional Development Fund (ERDF) via the Welsh Government.

References

1. Uecker M., Zhang S., Voit D., Karaus A., Merboldt K., Frahm J. Real-time MRI at a resolution of 20 ms. *NMR in Biomedicine*. 2010;23(8):986-994.

2. Zolgharni M, Ledger P. D., Griffiths H. Forward modelling of magnetic induction tomography: a sensitivity study for detecting haemorrhagic cerebral stroke. *Medical & Biological Engineering & Computing*. 2009;47:1301–1313.

3. Zolgharni M., Griffiths H., Ledger P. D. Frequency-difference MIT imaging of cerebral haemorrhage with a hemispherical coil array: numerical modelling. *Physiological Measurement*. 2010;31(8):S111.

4. Zolgharni M., Ledger P. D., Armitage D. W., Holder D. S., Griffiths H. Imaging cerebral haemorrhage with magnetic induction tomography: numerical modelling. *Physiological Measurement*. 2009;30(6):S187.

5. Soleimani M., Lionheart W. R. B. Absolute Conductivity Reconstruction in Magnetic Induction Tomography Using a Nonlinear Method. *IEEE Transactions on Medical Imaging*. 2006;25(12):1521-1530.

6. Brown B. M., Jais M. A variational approach to an electromagnetic inverse problem. *Inverse Problems*. 2011;27(4).

7. Knowles I. A variational algorithm for electrical impedance tomography. *Inverse Problems*. 1998;14(6).

8. Rodriguez A. A., Valli A. *Eddy Current Approximation of Maxwell Equations*. Springer-Verlag; 2010.

9. Ledger P. D., Zaglmayr S. hp-Finite element simulation of three-dimensional eddy current problems on multiply connected domains. *Computer Methods in Applied Mechanics and Engineering*. 2010;199(49):3386 - 3401.

10. Schöberl J., Zaglmayr S. High order Nédélec elements with local complete sequence properties. *The International Journal for Computation and Mathematics in Electrical and Electronic Engineering*. 2005;24(2):374-384.

11. Bangerth W., Heister T., Heltai L., et al. The deal.II library, Version 8.3. *Archive of Numerical Software*. 2016;4(100):1–11.

12. Kynch R. M., Ledger P. D. Resolving the sign conflict problem for hp-hexahedral Nédélec elements with application to eddy current problems. *Computers & Structures*. 2017;181:41-54.

13. Turcksin B., Kronbichler M., Bangerth W. WorkStream – A Design Pattern for Multicore-Enabled Finite Element Computations. *ACM Trans. Math. Softw.*. 2016;43(1):2:1–2:29.

14. Tournier P.-H., Bonazzoli M., Dolean V., et al. Numerical Modeling and High-Speed Parallel Computing. *IEEE Antennas and Propagation*. 2017;59(5):98–110.

15. Ammari H., Buffa A., Nédélec J. C. A Justification of Eddy Currents Model for the Maxwell Equations. *SIAM Journal of Applied Mathematics*. 2000;60:1805-1823.

16. Turner L. R., Davey K., Emson C. R. I., Miya K., Nakata T., Nicolas A. Problems and workshops for eddy current code comparison. *IEEE Transactions on Magnetics*. 1988;24(1):431-434.

17. Zaglmayr S. High Order Finite Elements for Electromagnetic Field Computation. PhD thesisInstitute for Numerical Mathematics, Johannes Kepler University, Linz, Austria2006.

18. Reinders J. *Intel Threading Building Blocks*. O'Reilly Media; 2010.

19. Kramer S. C., Hagemann J. SciPAL: Expression Templates and Composition Closure Objects for High Performance Computational Physics with CUDA and OpenMP. *ACM Trans. Parallel Comput.*. 2015;1(2):15:1–15:31.